Application-Level Isolation and Recovery with Solitude

Shvetank Jain, Fareha Shafique, Vladan Djeric, Ashvin Goel
Department of Electrical and Computer Engineering
University of Toronto

ABSTRACT

When computer systems are compromised by an attack, it is difficult to determine the precise extent of the damage caused by the attack because the state changes made by an attacker and those made by regular users can be closely intertwined. This problem occurs due to implicit sharing in operating systems, and it can be especially severe for persistent state. In particular, the file system provides a single namespace that when compromised can have cascading effects on the entire system, making intrusion analysis and recovery a time-consuming and error-prone process.

In this paper, we present Solitude, an application-level isolation and recovery system that is designed to both limit the effects of attacks and simplify the post-intrusion recovery process. Solitude uses a copy-on-write filesystem to provide a transparent, restricted privilege isolation environment for running untrusted applications, and it uses an explicit file sharing mechanism across the isolation environments that limits attack propagation without compromising functionality. Solitude provides two modes of recovery. If a sandboxed application proves to be untrustworthy, a course-grained recovery method allows easily removing the footprint of the software. However, if a user mistakenly moves malicious files to the trusted environment via explicit file sharing, then Solitude uses data dependency tracking to allow fine-grained recovery.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls, Invasive software (e.g., viruses, worms, Trojan horses); D.4.5 [**Reliability**]: Backup procedures, Fault-tolerance; E.5 [**Files**]: Backup/recovery

General Terms

Security, Reliability, Management

Keywords

Access control, Copy-on-write, File Systems, Recovery, Taint analysis, Transactional file systems

1. INTRODUCTION

Several research efforts in recent years have focused on analysis and recovery of compromised systems [12, 6, 9]. This problem is both real and hard: once a system is compromised, it is difficult to untangle the state changes made by an attacker, for instance the replacement of system binaries, from those made by normal users or administrators. While attempting recovery, an administrator is generally left with the choice of either confidently removing all attacker modifications or preserving all valid user activity, but not both.

We observe that a root cause of this problem is the implicit sharing that exists in modern operating systems. A principal example is the file system in which all users and processes share a single common namespace. Compromises that manage to make unauthorized updates to this namespace, for instance by replacing the commonly used UNIX `ps` command, can have cascading effects across the entire system. While operating systems provide separate address spaces to protect physical memory, comparable protection is limited for persistent state.

In this paper, we present Solitude, an application-level isolation and recovery system that is designed to both limit the effects of attacks and simplify the post-intrusion recovery process. At its core, Solitude provides a file-system based isolation environment for running untrusted applications. Each isolation environment is bound to its own file-system namespace, similar to a process address space, via a copy-on-write isolation file system called IFS. IFS offers a transparent view into the base (or regular) file system for reading operations, but any modifications made by the untrusted process or its children processes are confined to the separate namespace. If the user decides that the application is malicious or undesirable, the entire compromised IFS environment can be discarded without concern for the integrity of the base file system. This recovery method, being simple, is accessible to ordinary users.

A typical usage scenario of our system may involve running a peer-to-peer (P2P) client program within an IFS. For example, a user may download and install a photo editing application using the P2P client. The user knows that files on P2P networks are sometimes modified to include malicious components and thus installs the application in an IFS. This may be the same IFS as the P2P program or a new IFS, but in both cases, no changes are made to the base file system unless the user has explicitly authorized them. If the application exhibits unexpected or suspicious behaviour, the user can remove the program and its changes by discarding its IFS.

As with any isolation environment, there is a trade-off involved between the security provided by the IFS isolation environment, application-level functionality and ease-of-use. We rely on two mechanisms to resolve these concerns: support for restricted privileges for running server applications, and policies for explicitly sharing files between an IFS environment and the trusted base file system. Running programs with restricted privileges in IFS inhibits the spread and effectiveness of malware such as spyware, rootkits and memory-resident viruses that attempt privileged operations (e.g., loading kernel modules), and makes it harder to compromise the IFS isolation mechanism.

Support for file sharing policies enables rich system functionality and helps with ease of use. For example, a user may wish to use

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'08, April 1–4, 2008, Glasgow, Scotland, UK.
Copyright 2008 ACM 978-1-60558-013-5/08/04 ...$5.00.

image files created by an untrusted photo application in a presentation program running in the base file system or within another IFS. She may rely on a file sharing policy that allows the image directory to be shared with the base, or she may explicitly commit the images from IFS to the base. We expect that these sharing policies are specified when the application is installed or run for the first time by a user. While low-risk files may be shared with the base system for ease of use, commit-based sharing is more appropriate for most files because it delays synchronization with the base and thus allows handling errors in the sharing policy until commit is performed.

We realize that users rarely have perfect knowledge of a program's trustworthiness and that they may unwisely contaminate their base file system by designing insecure sharing policies or committing malicious data or applications. Solitude addresses this issue by using a taint propagation layer between *all* applications and the base file system. This layer tracks the effects of synchronizing files between the IFS and the base by recording how other applications access these files and the actions they undertake as a result. If a user decides that untrusted files or applications have made it into the base, she can use Solitude for intrusion analysis and recovery. Solitude helps produce a list of files and processes that may have been affected and provides the ability to intelligently rescue the system with fine-grained recovery tools.

Solitude makes three main contributions. First, it provides a file-system based restricted-privilege isolation environment that is reasonably easy to specify and can be used for both client- or server-side applications. Second, it uses an explicit data sharing model between an isolation environment and the base system that limits damage from vulnerable applications and improves accountability of persistent state changes. Finally, compared to our previous system [6], Solitude can track contamination more accurately and has significantly lower logging needs for system-level intrusion analysis and file-system recovery. Our evaluation shows that the Solitude model can be retrofitted in existing systems.

The rest of the paper describes our approach in more detail. Section 2 provides further motivation for the problem addressed in this work. Section 3.1 gives an overview of our system, describing the usage model and our threat model. Section 4 presents the architecture and the three main components of Solitude, the IFS isolation environment, the sharing policies and the taint propagation and recovery model. Section 5 describes the current status of the Solitude implementation, and Section 6 provides a detailed evaluation of our system. Section 7 describes related work, and Section 8 provides our conclusions.

2. MOTIVATION

As computer systems continue to be infiltrated and organizations lose customers and revenue due to attackers, the ability to do accurate analysis of attacks has become increasingly important. A serious problem faced by security professionals when doing such analysis is to distinguish attack activity from legitimate user activity. Previously, we designed a prototype analysis and recovery system called Taser [6] that securely audits all system-level activities on a target system so that these activities can be diagnosed at a later time. After an attack, Taser provides tools for analyzing the audit log and an investigator can run taint propagation on kernel objects such as processes and files to determine the set of attack related activities. Since Taser maintains a log of all file-system activities, it can revert the persistent changes made by an attack, e.g., removal of a trojan program.

The Taser approach is implemented within an operating system and requires *no* changes to existing applications, but it comes with a cost. First, the system needs to log all system-call activities since any activity could potentially be malicious. While such logging is not computationally intensive, it imposes heavy storage overhead and analysis can take a long time. Second, and more seriously, while taint propagation is a useful tracking technique, it raises problems in current operating systems because of implicit sharing. In particular, all users and processes share a single common file-system namespace. For instance, in most Unix systems, any user or application can write to the shared /tmp directory. Similarly, suppose that an attacker is able to modify a heavily shared file, such as the password file. Any future user logins would read this file and be marked tainted, and hence taint propagation would not be able to reliably distinguish between attack and legitimate user activity.

This implicit file-system sharing problem is exacerbated as users increasingly download and install software from untrusted sources on the Internet. Users are faced with the choice of either not downloading and running the application, or they risk compromising the integrity and the stability of the system. For instance, a downloaded media player application can have serious vulnerabilities that can allow attackers to attach malicious code and infect computers without the user's knowledge. Additionally, audio and video streams and downloads can be used to hijack or corrupt computers [33].

3. OVERVIEW

Rather than urge users to avoid downloading applications and media, we argue that typical desktop applications that are used on an almost constant basis such as web browsers, instant messengers, word processors, e-mail readers and media players as well as most server applications should *always* be run in isolated environments to limit the impact of attacks.

Since our primary focus is on simplifying analysis and recovery of persistent data, Solitude uses a file-system based isolation environment in which untrusted applications can only compromise their own namespace. The challenge arises when applications, running in different isolation environments, need to share data such as a media file that is downloaded in a browser and used by a media player. Solitude allows explicit sharing of files between an isolation environment and the trusted base system, but it marks any such shared files as tainted. Any use of these tainted files in the base is then logged and tracked, similar to Taser. However the source of tainting is limited to the explicitly shared files, and hence as explained later, Solitude not only requires much less logging than Taser but is expected to help determine attack-related activities more accurately. Below, we discuss the usage model of our system and our threat model.

3.1 Usage Model

Based on the notion that intrusions start with a network connection and then cascade into multiple system activities such as file accesses and outgoing connections, we envision that Solitude will be useful for various networked applications. These applications could either be client-side applications run by the same user or server-side applications (such as a web server, mail server, print server) run on a machine on behalf of the same set of users. Below, we describe some examples that represent usage models of our system.

Users increasingly run applications such as instant messaging to communicate and share data. These applications can be run in the IFS isolation environment, which by default allows read access to the user's file-system environment and isolates all updates. An *external* sharing specification describes how applications access the base file system, which is important because it allows retrofitting

the sharing policy onto existing applications. A service that knows, for example, that a messaging application writes chat logs to a certain directory can offer the user the option to preserve these logs while ensuring that all other persistent updates remain isolated. Similarly, with a mail client, the local mail directory and the mail-client configuration files could be explicitly shared with the base system while any other persistent data would be unshared.

On the server side, consider a web site that provides an on-line photo album service. The web server can be run in an isolation environment while configuring only the users' photo data to be shared with the base system. In this way, the persistent data that is important to users can be shared or committed to the base system, such as for archival or file search, but any updates made by the web server are unshared and cannot affect the rest of the site even if the web server is compromised.

The IFS isolation environment allows running multiple, related applications in a session. Consider a user that uses a peer-to-peer application to download files. After downloading the file to a standard location, the user can run a viewer application within the same session or mark the standard download location as explicitly shared and use the viewer in the base environment. All other updates by the peer-to-peer application are unshared and could be easily discarded after session termination. Note that isolation environments are persistent in the sense that the IFS state is preserved across multiple invocations of the application.

Administrators can also choose to use IFS environments for certain low-privilege users. For example, IFS can be used to ensure that anonymous FTP users cannot affect the base file system, and to isolate directories that are shared across users such as the Unix `/tmp` directory that has been the source of several exploits.

3.2 Threat Model

A malicious application can damage the integrity, confidentiality and availability of a system. Solitude strives to preserve the integrity of the system in the presence of untrusted networked applications. The integrity of a system can be compromised by unauthorized modification of files and misuse of capabilities. For example, an application can delete important binaries or load a rootkit leaving the system in a corrupted state. Solitude attempts to contain the damage caused by such operations using two methods: 1) limiting access privileges to the trusted file system by isolating the file modifications made by an untrusted application to a separate namespace and providing fine-grained access control to the trusted file system, and 2) limiting system-wide capabilities available to an untrusted application in keeping with the least privilege principle. These methods limit attack propagation in the file system by denying access to the files that should not be modified by the malicious application, and restrict an application from causing harm by misusing its capabilities, which are far fewer than the all powerful root user in Unix systems. An untrusted application may also communicate with other processes via means other than the filesystem, such as IPC or covert channels. Solitude tracks accesses to the trusted file system and IPC communication via a taint tracking mechanism that can be used for post-intrusion analysis. However, Solitude does not track covert channels, for example, communication via the external network.

In addition, a malicious application may trick a user into granting additional privileges or may misuse its privileges. Solitude aims to counter this threat by providing a system-wide recovery mechanism, allowing users to recover their system. A malicious program can also breach confidential information on the system by leaking it to the outside world. Such techniques, e.g., employed by spyware programs, attack user privacy. Although not a fundamental objective of Solitude, these risks can be reduced by denying read access to sensitive information on the disk. A malicious application can also reduce system availability. In this case, Solitude does not provide any additional protection, but is no worse than the original system.

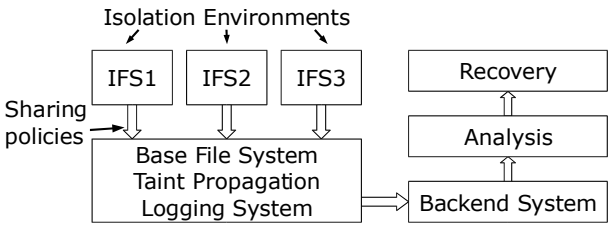

Figure 1: The Solitude architecture

4. APPROACH

Solitude provides a copy-on-write file-system based isolation environment for running untrusted applications, and it uses an explicit file sharing mechanism that limits attack propagation without compromising application functionality. The Solitude architecture is shown in Figure 1. It consists of three main components, the IFS isolation environment, the sharing policies, and the taint propagation, logging and recovery system. We describe these components below.

4.1 IFS Isolation Environment

Solitude allows running an untrusted application in an isolation environment called IFS that provides the application with a transparent view of the base file system, but it restricts any file-system changes with a default copy-on-write policy. This policy can then be refined with explicit sharing policies that allow synchronizing the base file system with the IFS. Below, we first describe our motivation for using this isolation model and then present the model.

4.1.1 Copy-on-Write

There are several reasons that motivated a copy-on-write based isolation environment. First, the basic file-system recovery method is simple: if at any point the user decides that the software may be malicious, they can discard the entire IFS environment without concern for the integrity of the base file system. Second, explicit sharing of file updates limits attack propagation across IFS environments and hence reduces the effort involved in overall post-intrusion analysis and recovery. Finally, copy-on-write enables read sharing between the base and the IFS, and we do not require explicit read sharing because such operations are far more common and thus configuring them correctly would be challenging. However, as discussed later, the Solitude sharing policies can be configured to deny access to sensitive base files in the IFS environment.

With copy-on-write isolation, any malware that attempts to conceal its presence by disabling security software or by installing rootkits will fail because the isolation mechanism safeguards the integrity of programs in the base system. Similarly, spyware that embeds itself in browsers will be ineffective when run inside an IFS that is not specifically used for web browsing. Copy-on-write isolation can also help stop the spread of worms and viruses that propagate across network filesystems because IFS makes local copies of these files. Finally, it can aid with malware detection. For example, a spyware detector could be periodically run in IFS, and copy-on-write would allow easier monitoring of malware activity.

Capability	Parameters
Fcap	FilePath [owner.group] [perm]
CAP	[AppPath] CAP_SET

Table 1: IFS capability model

4.1.2 Isolation Model

We create an isolation environment by mounting the copy-on-write IFS file system at a predefined mount point in the base file system and starting an untrusted application within a `chroot` jail rooted at the mount point. With the default policy, the application has read access to the entire base file system but write access is limited to IFS. We ensure that applications in the IFS environment execute with privileges more restricted than if the same applications were run in the base environment. For example, if an application cannot access a root-owned file in base, then it will not be allowed to access the file in IFS also.

To strengthen this isolation mechanism, we have incorporated the Vserver secure chroot barrier [26] in IFS. This barrier uses a special flag on the parent directory of the isolation environment to prevent chroot escape. However, even with this mechanism, techniques for escaping chroot jails are known and have led to best practices for using them [4]. The most important of these rules is to disallow all-powerful `root` privileges in a jail, which makes it significantly harder to escape the jail. Unfortunately, this method limits its functionality by disallowing setuid programs and server-side applications that, for example, may require access to privileged ports. Setuid programs are appealing targets for Solitude isolation. They are frequent targets of attacks because they provide a direct path to complete control over the system. Sendmail is a typical example of a setuid application – it uses its root powers to temporarily impersonate other users to deliver mail to their inboxes.

IFS restricts the privileges of root or setuid applications by enhancing the capability system available in Linux [15]. Each IFS environment can specify capabilities that are then enabled in the environment. For example, a web server IFS environment would allow opening privileged ports. We chose Linux capabilities because they are relatively easy to specify. However, they are coarse grained, and in particular, file related capabilities apply to the entire file system. For instance, a program running with the CAP_DAC_OVERRIDE capability can override *all* file access restrictions.

Instead of allowing such powerful file capabilities, IFS allows per-file or directory privileges that allow overriding file access restrictions in the base. This approach may seem cumbersome, but our results show that in practice systems configure the discretionary access control permissions for most files "almost" correctly so that few overrides are needed. Hence privileged applications typically require few per-file privileges and can be run correctly without full-root privileges. For example, consider a web server such as Apache2 running with restricted privileges in an IFS environment. The application does not run as the root user and is only given the capability to bind to a privileged port. As a result, it does not have permission to access some files that are root owned, such as its error log and access log. Therefore, it must be given per-file privileges for these files. However, it can access most of its other files, such as data and configuration files, without requiring additional privileges.

Table 1 shows the specification of the IFS capability model. These capabilities are specified for each IFS environment. The `Fcap` file capability allows *overriding* the file ownership or permissions on the file (or directory) specified by `FilePath`. It provides a fine-grained version of the Linux CAP_DAC_OVERRIDE capability, and it applies to all programs run within an IFS. The CAP capability

```
Application /usr/sbin/apache2 www-data.www-data
Fcap /var/run/apache2.pid www-data.www-data
Fcap /var/log/apache2/error.log www-data.www-data
Fcap /var/log/apache2/access.log www-data.www-...
CAP  net_bind_service
Wcommit /var/log/apache2/error.log
Wcommit /var/log/apache2/access.log
```

Figure 2: Policy for the Apache web server

applies to the application specified by `AppPath`. When `AppPath` is not specified, it applies to the top-level application. This capability is enforced when an application starts executing, e.g., on a Unix `execve` system call, and the CAP_SET parameter is a list of capabilities (such as accessing privileged ports) allowed to the application. IFS restricts certain capabilities such as create or remove mount points and accesses to raw devices, that may allow applications to escape its isolation environment.

IFS capabilities are specified in a per-IFS policy file. Figure 2 shows a policy file for the Apache web server. This file is saved outside IFS in the base system and can specify 1) the principal the application should run as inside IFS, 2) the IFS capabilities, and 3) any of Solitude's explicit sharing policies (described in the next section). In Figure 2, Apache is run as the `www-data` user in an IFS environment and the ownership of files with the `Fcap` capability is set to `www-data` in the IFS (not in the base) environment. This capability together with the `net_bind_service` capability for accessing a privileged port allows running Apache in an IFS environment with no other privileges. Section 6.2 shows that it is easy to specify these capabilities for our other targeted applications.

When an application running within an IFS starts a child application by executing the `execve` system call, the `CAP` capability of the child application is *exactly* the set specified by the `CAP_SET` parameter for that application, and as a result, the child does not inherit any capabilities from the parent application and setuid applications have no additional privileges in IFS. Furthermore, each capability is specified for a given IFS and is not system wide. We also ensure that files that have been copied into IFS via copy-on-write do not run with any CAP capabilities. This is to ensure that a vulnerable IFS application cannot be hijacked into executing a privileged application that has been modified.

4.2 Sharing Policies

Solitude isolates the persistent changes made by applications by using copy-on-write as its basic isolation model, but it allows refining this model with policies that enable explicit sharing of specific files and directories between an IFS and the base system. This explicit file sharing model is based on the idea that sharing of files across applications is rare. We evaluate this hypothesis in Section 6.1.

The sharing policy language is designed for simplicity and intuitiveness similar to the IFS capability model. In particular, there is no support for sharing between two IFS environments directly, which provides finer-grained sharing but would complicate the language design and specification. Any such sharing must be performed via the base system. The sharing policy specifies three sharing modes for reading and three modes for writing (although a few combinations are not meaningful and mentioned below). These modes are shown in Table 2. All the sharing modes apply to a file or directory and are subject to the access control restrictions of the base unless these restrictions are overridden by the `Fcap` capabilities. Below we describe the different modes and show how they can be used to support the IFS usage models described in Section 3.1.

Read/Write Mode	Description
Rshare	Allow file read sharing (default)
Rsnapshot	Allow file read from a snapshot
Rdeny	Deny file read
Wisolate	Deny file write to base (default)
Wcommit	Allow file write during commit
Wshare	Allow file write sharing

Table 2: IFS sharing modes

Rshare specifies the default read policy in which IFS applications transparently read from the base system until they make an update to the object. At this point, the data is propagated to the IFS and read from there (one way copy-on-write). The read-share policy protects the base system from any persistent changes made by a compromised application running in an IFS and also allows easily rolling back these changes.

Rsnapshot specifies copy-on-write in both directions, i.e. the IFS makes a snapshot of the base immediately upon start-up, rendering both the IFS and the base oblivious to any subsequent changes made in the other. A read snapshot ensures that software updates in the base do not affect the isolation environment. However, it also disables critical updates, such as security updates for a web server vulnerability, from becoming available in the isolation environment. The isolation environment must be shutdown and reinitialized to access these updates. At shutdown, updates in IFS can be discarded or commit shared as described below.

Rdeny hides specified base files or directories from the IFS environment and therefore guards against information leaks and addresses information privacy concerns. The read-deny policy is likely to be used in systems where the isolation environments are used to run risky applications susceptible to subversion such as certain network servers. A read deny overrides any write sharing policy.

Wisolate specifies the default write-isolate policy and confines all writes permanently to the IFS.

Wcommit specifies that changes are confined to IFS but may be eventually committed to the base system. Commit sharing delays synchronizing updates until an explicit commit is issued. During commit, IFS updates to files and directories specified via Wcommit are applied to the base system atomically and are then discarded from IFS. Each commit is associated with a Commit ID that is used during recovery, as described later in Section 4.3. This sharing mode provides a time period (until commit) during which updates can be discarded and is suitable when sharing data loosely or occasionally with the base.

Commits are initiated by starting a commit process on a particular IFS. The commit process is described in more detail in Section 5.3. Commits can only be invoked by the IFS owner in the base system, ensuring that malicious applications running in the isolation environment cannot commit persistent data. An alternative that is currently not supported in Solitude, is to allow commit sharing from the isolation environment after explicit user authentication similar to the use of the `sudo` program in Unix systems.

Various usage models described in Section 3.1 can benefit from commit sharing. For example, the session logs of instant messaging applications could be commit shared with the base system periodically. Similar, for the photo application described earlier, the photo album directory could be commit shared with the base system for archival or for viewing by another application such as file viewer.

```
Application /usr/bin/firefox
Wcommit /home/djeric/downloads/
Wcommit /home/djeric/.mozilla/
Rdeny /home/djeric/private_info.txt
```

Figure 3: An application policy file with a write-commit and a read-deny policy

Solitude allows concurrent updates to occur in the base and the IFS and hence persistent state in the two file systems can diverge over time. An object modified in IFS can be committed successfully when the corresponding object in the base file system has not been modified after it was first copied into IFS. This criteria ensures that the commit operation is atomic and equivalent to an object being accessed and modified at commit time [30].

When the commit criteria is not met, updates can diverge and the commit is said to be conflicting. Such conflicts can be automatically resolved for directories since their semantics are known. We expect that sharing and hence conflicts across applications described in our usage models will be rare. If conflicts are common, then either the conflicting applications should be run within the same IFS, or the write sharing method described below is more appropriate.

Although Solitude is designed to run untrusted applications indefinitely within the IFS environment, it also supports committing an entire application that the user deems trustworthy after evaluation.

Wshare indicates immediate write-sharing of changes from the IFS to the base file system and is useful for files known to be of low risk. Write sharing is also used for special files such as device files that typically do not satisfy file semantics (i.e., reads provide the same data as a previous write to the file) and also do not provide persistent data. These files must be explicitly specified as shared to allow updates. For example, many Unix programs write to the `/dev/null` device file and terminal programs write to the `/dev/ptmx` pseudo terminal device. A write-shared file must be in read-shared mode.

Figure 3 shows a sample policy snippet for sharing files and directories. It specifies that any changes made by the Firefox application are permanently confined to its IFS (the default policy), with the exception of the `downloads` and the application profile directory which can be explicitly committed by the user to the base file system. Firefox is allowed to read all the files in the base except for the file `private_info.txt`, presumably because it contains sensitive information.

Policy rules are applied recursively to a directory's files and subdirectories. A more specific rule, such as one that applies to a file in a sub-directory, overrides a more "general" rule, i.e. one that applies to a higher-level directory and its sub-directories. This allows users to specify a rule covering a directory and its contents, with exceptions for some sub-directories and individual files.

The intended authors of the policy files containing the IFS capability and sharing specification are companies that provide the OS distribution or user communities. The policy language is simple and we expect that system administrators will be able to easily modify the policy files to suit their environment. For certain applications, a default read-only policy should be satisfactory. In such a scenario, the damage is restricted to information leakage. The user can mark sensitive files with RDeny, thereby avoiding their leakage.

Innocuous applications, when hindered, can inform and request the user to grant certain file access permissions and capabilities to

Operation	Propagation Rules
File and directory read operations, execute file	Tainted file → Taint process
File and directory modification operations	Tainted process → Taint file
Create child process	Tainted process→ Taint child process

Table 3: Taint propagation rules

function correctly. At this point, the user must judiciously grant these privileges, based on the files that can be modified as well as the capabilities that can misused. A mistake in his assessment will require system-wide recovery as described below.

4.3 Taint Propagation and Recovery

The sharing policies described above enable collaboration between applications running in IFS and base, or between different IFS contexts via the base. Without support for such sharing, an increasing number of applications would be run in the same isolation environment, negating the benefits of isolating the applications. However, the sharing policies could be poorly designed, potentially leading to contamination of the base file system either via commit or write sharing of malicious data or applications. Solitude addresses this issue by tracking how other applications access files that are committed or write shared, and then using a taint propagation method to log their resulting actions. If untrusted files reach the base, Solitude uses a modified version of our Taser system [6] for analysis and fine-grained recovery of the base system. Below, we describe the Solitude taint propagation method, the logging system and our offline recovery method.

4.3.1 Taint Propagation

Each IFS environment in Solitude conceptually has an associated *IFS monitor process* running in the base that performs all file operations on behalf of IFS processes in that environment. This process accesses a file in the base or the IFS environment based on the file sharing mode, and synchronizes files from the IFS to the base during a commit. Solitude marks this monitor process as tainted since it operates on behalf of untrusted IFS processes. Then the taint propagation algorithm tracks modifications to the base file system that depend on this process. The taint propagation rules are simple and shown in Table 3. These rules operate on kernel objects such as processes, files and directories that can either be untainted or tainted. The rules 1) taint a process when a process reads or executes a tainted file, 2) taint a file when a tainted process modifies the file, and 3) taint children processes of a tainted process. These rules can taint any processes running in the base environment or any files in the base file system, but they ignore IFS processes or files, because our goal is to recover the base system after an attack.

The Solitude tainting algorithm, operating at the granularity of kernel objects, is coarse grained compared to instruction-level data-flow analysis [29, 18, 3]. We choose to use a coarse tainting method for two reasons. First, data-flow techniques are vulnerable to attacks caused by implicit or control-based information flows within a program [24]. Our algorithm taints at the process level and thus does not suffer from this problem. Second, data-flow techniques generally have a large overhead and are thus not used during normal operation, while our algorithm has low overhead and can be used in real time. The main drawback of coarse-grained tainting is that it can introduce a large number of false sharing dependencies. However, we expect that file sharing will be common mainly within IFS environments (which are ignored by the tainting algorithm), and our explicit sharing policies will limit false dependencies in the trusted base environment.

4.3.2 Logging System

The taint propagation algorithm uses a single bit to taint base processes or files according to the rules shown in Table 3. However, this approach only allows determining the entire set of processes or files that were tainted by any isolation environment. Solitude allows finer-grained recovery at the commit level by logging two kinds of operations. First, it logs all operations in which the source object in any propagation rule shown in Table 3 is tainted. For example, in the first rule, it logs a file read operation and the process reading the file when the file is tainted in a *tainted operation log*. Second, Solitude uses a separate privileged *commit process* to generate a *commit log* that stores an IFS ID, a commit ID (that is incremented per IFS commit), and the set of committed files on each commit. The commit process taints itself, and hence its operations are logged in the tainted operation log. This log and the commit log are sent to a backend system shown in Figure 1, which allows commit-level recovery as described below. The logs are stored and analyzed on a separate system so that they cannot be easily destroyed.

Our previous Taser system [6] assumed that all file-system operations were untrusted and hence logged all these and related operations. In contrast, the Solitude threat model assumes that untrusted applications are executed in IFS environments and hence only operations in IFS environments and tainted operations in the base are untrusted. After an attack, copy-on-write based IFS environments can simply be discarded. As a result, Solitude does not need to log any operations in IFS and only logs operations on tainted base objects for fine-grained recovery. This approach can reduce logging significantly compared to Taser.

4.3.3 Recovery

Recovery in Solitude can be performed on the backend system at a per-IFS, per-commit granularity. If a malicious file is committed to the base, the user specifies the IFS from which the file was committed and a commit ID (described earlier) as a starting point for recovery. Our analysis tools, previously developed in Taser, help determine the IFS and the rollback commit ID. For example, each commit stores a commit time and the set of committed files. If the file is known to be committed at some approximate time, then the closest previous commit ID is chosen.

The recovery process can generate the tainting dependency graph for any given commit. It starts by tainting the corresponding commit process and re-running the same tainting algorithm described earlier. This is possible because all tainted operations are logged to the backend. Our use of the IFS monitor process (see Section 4.3.1) ensures that if the tainted operations of an IFS are read by another IFS, then the recovery process will track the operations of the second IFS also. Furthermore, since an IFS environment will often read files after they are committed, all subsequent write-share and commit operations by the same IFS will also become tainted. As a result, recovery for the explicit shared operations in Solitude would be performed at the granularity of IFS environments. This approach represents a trade-off between the granularity and scalability of recovery. Previously, we used Taser to perform rollback at a finer process-level granularity, but that required logging all file-related

operations in the system which limited the scalability of the system, and it also required more detailed analysis to determine the starting point for the tainting algorithm. Since IFS environments are typically used to run closely-related applications, we believe that IFS-level recovery is an acceptable trade-off.

Once the set of files that depends on a particular commit ID has been generated, these files should be manually inspected to ensure the correctness of recovery. Then the files can be rolled back to an untainted state by using the unmodified selective redo algorithm in Taser [6].[1] The rollback uses a snapshot of a file taken when the file was first tainted by any commit and then replays all subsequent modifications (which were logged because the file was tainted) until the file reaches a state just before the rollback commit ID. The recovery system uses a simpler, more efficient undo algorithm for reverting tainted directory operations, and then generates a self-contained script that can be used to recover the base file system.

5. IMPLEMENTATION

Solitude consists of three main components, the IFS isolation environment for running untrusted applications, the explicit sharing policies for limiting attack propagation and the tainting system for recovering from malicious shared files. Next, we describe our implementation of these components.

5.1 IFS Isolation Environment

The basic isolation mechanism in IFS is a copy-on-write file system. For ease of implementation, we have developed a user-level prototype of this file system using FUSE [31] running on the Linux kernel. FUSE intercepts operations at the virtual file system (VFS) layer, so that applications do not have to be modified to work with FUSE file systems, and calls wrapper functions in a user-level process that performs all file system operations on behalf of the applications running in each IFS environment. This process, conveniently the IFS monitor process described in Section 4.3.1, implements copy-on-write by redirecting operations to the base or IFS layer. This implementation runs on the Linux `ext3` file system but is mostly independent of the base file system.

5.1.1 Implementation of Copy-on-Write

IFS implements copy-on-write at the file-system level. The implementation for files is straight-forward – files are copied to the IFS whenever file data or attributes are modified. An IFS directory is an overlay that only contains files or sub-directories in IFS. It is created when 1) a base file (or sub-directory) within the directory is modified, or 2) an IFS file (or sub-directory) within the directory needs to be created. For example, when a base file is modified or a file is created, IFS directories are created for all ancestor directories of the file.

The implementation must handle three main issues. First, a create-delete ambiguity is introduced when a file that was copied from base is removed in IFS. The implementation ensures correct copy-on-write operation by recording all file deletions and the deletion time. Second, it records the time when a file is first created in IFS. This time-stamp is used during commit to detect file content conflicts, which occur when this time is earlier than the base file modification time. Finally, hard links in Unix file systems, which allow a single file to have more than one name create several complications. For example, consider a file with two hard links that exists in the base and is updated in IFS using one name. Later, it is accessed in IFS with its second name. In this case, IFS needs to map this name to the IFS version of the file. The implementation uses two mapping tables, one from base inode of a file to the corresponding IFS inode, and another from IFS inode to the file name and the IFS inode of its parent. These tables allow mapping a pathname in the base to a pathname in the IFS. IFS must also perform reference counting to ensure that an IFS file is not removed until all names of the file in IFS and base are removed. Additional details are available in our technical report [27].

[1]Taser also provides tools that help a user analyze and modify the set of files that need to be reverted.

5.1.2 Implementation of Isolation Model

When an IFS isolation environment is started, the top-level IFS application, by default, assumes the ID of the user invoking the IFS environment. However, if a user is specified in the policy file (e.g., see the first line of the Web server policy file shown in Figure 2) and this file is owned by user `root` and not readable or writable by others, then the policy file user ID is used. Similarly, applications can only acquire the capabilities shown in Table 1 if the corresponding policy file is `root` owned and non-world readable or writable.

The capabilities in Table 1 are implemented using two methods: the `Fcap` capability is enforced by the IFS monitor process, while the per-application `CAP` capability is implemented by modifying Forensix [5], a kernel-level system-call interception facility. At IFS start-up, this capability is passed to the kernel, stored in a per-IFS kernel data structure and enforced during the `exec` system call. Unlike the current Linux security model in which applications that require any privileges are run with all privileges (as root), our implementation ensures that only the privileges needed by any application are given to it, thereby restricting the Linux security model. Currently, we require manual specification of IFS capabilities for an application. We plan to develop a tool that will help simplify this process.

5.2 Sharing Policies

The Solitude sharing policies are implemented within the IFS monitor process. The default policy is copy-on-write, i.e. read share and write isolate, with a few exceptions such as write sharing for some `/dev` devices. Currently, we are in the process of implementing the `Rsnapshot` mode in Solitude. This mode requires integration with a standard file-system versioning mechanism [37, 19, 1], and is especially useful if the user plans to run multiple versions of an application in different IFS environments.

While our sharing policy language is simple by design, we find that determining the correct policy for a large application can take time. For example, determining the full set of files used by an application and determining the correct policy for each sharing scenario can be challenging even for individuals with intimate knowledge of the internals of an application. Even when the correct policy has been supplied externally such as by a community of users, individual system administrators may still wish to customize policy files to their specific system configurations and specific needs.

Accordingly, we have created a simple tool that profiles an application's file I/O behavior and produces a list of files that are created, deleted, read or written by the application. This output provides a good starting point for deciding the set of files that need to be shared. In Section 6, we describe our experiences with writing sharing policies for several popular client- and server-side applications.

5.3 Taint Propagation and Recovery

We have implemented taint propagation and the logging system (for recovery) in the kernel of the target system with Forensix. The taint propagation algorithm starts when a user commits files to the base file system or when files are write shared. File commit is im-

plemented as a separate commit process, although it is logically a part of the IFS monitor process since it also enforces the sharing policies. The commit process uses redo logging to ensure atomicity. This process first explicitly taints itself (similar to the IFS monitor process which is always tainted), creates a commit log of operations to be performed (e.g., creating or synchronizing a file in the base), sends the commit log to the backend with a new system call called `solitude_commit`, and then performs the operations. Tainting the commit process taints all its file operations, which are then automatically appended to the tainted operation log and sent to the backend system. All IFS updates to write-shared files between two explicit commits are associated with the subsequent commit, which provides a logical grouping of the various modifications to the base file system to discrete commit points for the purposes of recovery.

The taint propagation algorithm maintains the taint status of base processes and files by simply following the rules shown in Table 3. File taints are maintained persistently across reboot. The logging system initially sends a snapshot of the base file system namespace to the backend. After that, it logs all base namespace modifications to the backend so that the backend recovery process can recreate a consistent view of the base file system. When a file is first tainted, a snapshot of its pre-tainted contents are sent to the backend. After that, all content updates to the tainted file are logged to the backend. The backend can then rollback a file to its pre-tainted state and apply updates until the time associated with a specific rollback commit ID.

6. EVALUATION

Our evaluation of Solitude focuses on the effort involved in configuring sharing policies and how well the system limits the spread of contamination from the untrusted IFS environments. We start by evaluating sharing patterns among existing applications to gauge the complexity of isolating various classes of applications. Second, we describe the effort involved in configuring the sharing policies and capabilities for applications run within IFS environments to determine the usability of the system. Third, we measure the contamination that occurs due to explicitly shared files and the storage requirements of tracking and logging the actions of contaminated processes. Finally, we evaluate the performance overhead of Solitude.

6.1 Measurement of Sharing

Our first study is designed to evaluate file- and IPC-based sharing patterns among existing applications run in Linux environments. Our hypothesis is that *both* file and IPC-based communication is limited among applications that are targeted for IFS environments (e.g., network applications) and hence configuring explicit sharing for these applications is a viable option. We test this hypothesis for both client and server environments by using Forensix [5], a system that logs all system calls and provides MySQL-based tools that help with analysis of past system behavior. The client system is one of the authors' machines and it runs Ubuntu Linux 2.6.15. The server also runs the same OS and provides web server (with a php/mysql backend), imap, webmail, postfix, NFS, VNC, dhcp, tftp and sshd services to a cluster of 128 machines with roughly 10-15 active users. We present results for any potential sharing that occurs among applications based on file and IPC-based communication.

Table 4 shows the results for write-write sharing of files by more than one program in both a client and a server environment. This table indicates that sharing is relatively uncommon compared to the total number of file accesses on the client side. On the server side, the majority of the sharing occurs due to a set of related mail programs shown in the first row of Table 5. This table shows the shared files, the programs that accessed them, and the type of sharing that would be needed to support these applications in a Solitude environment. The first column of each row shows the number of shared files in the client and the server study in parenthesis. The Unshared files can be accessed in an IFS environment with no additional policies. The Write-shared files require a corresponding write-shared policy. Some files, such as log files, may either be in Unshared mode when accessed from IFS, or could be directly accessed by programs in the base environment. Also, certain programs, typically used for system administration, will mainly be run in the base. Files in home directories may be accessed within IFS or base depending on the types of programs being used. For example, a development IFS could be used for accessing files from a remote repository and editing and compiling these files. Similarly, other networked applications, such as web browsers would be run in their own IFS and their configuration files could be periodically committed to the base home directory.

	Client	Server
Experiment dates	Jul 26-Aug 23	Jul 25-Aug 13
Experiment time	29 days	20 days
Files written	30353	151856
Write shared files	173	88154

Table 4: File sharing statistics

We also performed an IPC study using the same experimental data to determine whether an explicit IPC specification is reasonable across IFS and base environments. Common IPC mechanisms in Unix systems include FIFO, Unix domain sockets, shared memory and local INET (TCP, UDP) sockets. The first three mechanisms have unnamed and named counterparts. The unnamed mechanisms work for related programs in a process hierarchy and are allowed within an IFS but disallowed across IFS and base in Solitude. For named communication, our study showed that there was no shared memory communication, and a very small set of applications used FIFO and Unix domain sockets during the course of the experiment.

Based on this initial result, we disabled these IPC mechanisms across IFS and base and different IFS environments to avoid implicit sharing, and re-ran the few applications using the IPC mechanisms. Surprisingly, we found that they still worked correctly. For example, Gnome applications use Unix sockets to communicate with the Gnome application-configuration registry. Our experiments show that disabling Unix domain sockets has no effect on these applications running in an IFS because they start another configuration daemon in the IFS. The configuration data is stored in a file hierarchy, and hence we were able to use commit sharing to synchronize the application configuration data in the IFS with the base (if desired by the user).

For local INET sockets, we saw no UDP based communication during the entire experiment. We saw local TCP connections to three services, the printing server, X server and the ssh server. The printing server should be run in an IFS, but the other servers provide basic services (desktop environment and remote access) and would need to be run in the base. In all these cases, these services would need to be shared with many IFS environments. Even so, these results are promising because the total number of such services is small, and hence we plan to incorporate explicit IPC specification in Solitude.

Shared files (client, server)	Programs	Type of sharing
Files in /var/spool/postfix (0, 85927)	Postfix, smtp, local, cleanup	Unshared
Files in home directories (30, 1639)	Compilation, etc.	Unshared, Commit In base
Files in /tmp (122, 553)	Cron, tex, other programs	Unshared
Files in /var/mail, /var/mail/$USER.lock (0, 10)	Procmail, imap, mail clients	Write shared
Device files, /dev/null, /dev/ptmx, /dev/ttyXX, /dev/pts/0 (5, 3)	Numerous programs	Write Shared
Log files, /var/log/wtmp, /var/run/utmp, /var/log/lastlog, .xsession-errors (5, 6)	gnome-pty-helper, xterm, sshd, sessreg	Unshared, In base
Libraries, /usr/local/lib/libfuse.a, /usr/local/lib/libulockmgr.a (2, 0)	install, ranlib	In base
Files in /var/lib/belocs, /var/lib/texmf/ls-R, /var/lib/dpkg/lock, /var/cache/debconf/ (9, 16)	locale-gen, synaptic, apt-get, gnome-session, gnome-panel	In base

Table 5: File sharing on a client and a server system

6.2 Sharing Policies

In this section, we discuss the usability of our system by describing examples of sharing and capability policies for various classes of client and server-side applications suited for IFS environments.

6.2.1 Client Applications

We wrote and tested policies for a web browser (`firefox`), instant messenger (`gaim`), mail client (`thunderbird`), a file-sharing client (`limewire`) and an audio player (`xmms`). These applications are representative of a large class of networked applications used for downloading data and executables on the client side. We used a combination of the copy-on-write IFS environment and our profiling tool to determine the files accessed by these applications.

These applications do not require any write-shared files except some device files. In all cases, the default specification for all these applications allows committing the application profile directory in the user's home directory to ensure that the application profile is safe even if the application is subverted and its IFS environment needs to be discarded. Conflicts during commit are unlikely, since we do not expect that an application's profile directory will be modified by other base applications. The user may also specify that the downloads directory of some of these applications (e.g., `~/Share` for `limewire`) can be committed. The mail client policy is similar when using the POP or IMAP protocol. However, when mail is delivered locally, the mail INBOX folder must be write shared to enable sharing with the mail transfer agent (e.g., `postfix`) IFS and all mail folders must be write shared when using a mail delivery agent (e.g, `procmail`). Figure 3 showed a sample policy for `firefox`. The other applications have similar policy files.

While the default policy for all these applications requires three or fewer lines, the user may choose to use a more fine-grained specification. For example, the user may write- or commit-share only the `logs` directory of the `gaim` application instead of the entire application profile directory. Similarly, the user may choose to commit only the bookmarks file in `firefox` and specific extensions that are known to not be malicious [13]. All policies could be further refined by users to protect the privacy of specific files and directories using the read-deny policy. Finally, if these applications were IFS aware, these policies could be set up based on user input when the user runs the application for the first time.

6.2.2 Server applications

We wrote and tested policies for server applications like a web server (`apache2`), a web server with a php-based photo application (`gallery`), a mail server (`postfix` and `procmail`), an IMAP server (Dovecot), a DHCP server (dhcp3), a print server (Cupsd), an SVN server (Svnserve), and an ftp server (`vsftpd`) based on most of the services running on our cluster server (see Section 6.1). We used our profiling tool to derive the file sharing policies for these applications.

```
Application /usr/sbin/postfix root.root
Wshare /var/mail/
Fcap /var/spool/postfix/pid/ root.root
Fcap /var/spool/postfix/private perm=00750
CAP /usr/lib/postfix/master net_bind_service
setgid setuid
CAP /usr/lib/postfix/pickup setgid setuid
...
```

Figure 4: Policy for the Postfix MTA

The basic web server policy is shown in Figure 2. This policy only allows committing the server log files for safe keeping. The `Fcap` capabilities are needed so that `apache2` can access the relevant files when running as the `www-data` user (these files are owned and readable only by the root user in the base environment). We also downloaded and ran the `gallery` application [17] within the web server running in an IFS. This application stores albums, pictures, album users, etc. in the `/var/www/albums` directory. We added a single commit line in the `apache2` policy file for this directory because it stores important user data. Any other operation performed by `gallery` or `apache2` is confined to the IFS.

The `postfix` policy shown in Figure 4 allows write sharing of the `/var/mail` folder so that mail clients running in a different IFS can access these folders. Postfix runs its main process as user `root` but it has various processes running as user `postfix`. It needs file capabilities because the `root` user has no special privileges in IFS. The `master` proces needs `net_bind_service` to bind to privileged port 25 and `setuid` and `setgid` because it runs processes as user `postfix` and changes its identity to each user that receives mail. Seven other processes are forked by the `master` process and start executing as the root user. They require setuid and setgid to later switch to the postfix user (only one is shown in the figure). The policy files for the rest of the applications are all simpler than the postfix policy and not presented here.

6.3 Taint Propagation and Logging

In this section, we measure the contamination that can occur due to explicitly shared or committed files and the storage requirements of tracking and logging the actions of processes contaminated by the taint propagation rules shown in Table 3. This data is hard to collect because it requires attacks on real user systems. A hon-

	Client	Server
Experiment time	10 days	14 days
Total log size	30.8GB	26.1GB
Total # of events	454.4 M	151.8 M
Namespace events	0.2 M (.04%)	3.5 M (2.3%)
Total files	276,118	3,315,437

Table 6: Forensix logging statistics

	Tainted files	Logged Events
Client		
acroread	2	0.3 million (.07%)
firefox	107	0.4 million (.09%)
amsn	134	0.4 million (.09%)
thunderbird	174	1.7 million (.37%)
nautilus	198	1.4 million (.31%)
gedit	222	10.0 million (2.2%)
Server		
svnserve	65	3.6 million (2.4%)
apache2	5	3.7 million (2.4%)
dovecot/imap	35	4.7 million (3.1%)
mysqld	15	3.7 million (2.4%)
pine	20	4.3 million (2.8%)
procmail	38	4.8 million (3.2%)

Table 7: Taint propagation and logging

eypot can be used to detect attack activity, but it may cause little contamination because it has no real users. Instead, we use the data collected in the user study described in Section 6.1 to provide an estimate of the level of contamination that may occur and the logging requirements during normal system activity.

For this experiment, we tainted all invocations of an application and measured the number of tainted files in the system and the amount of logging that would have occurred over the course of 10-14 days. This experiment was performed entirely in the backend system, and simulates the case of an application being tested in IFS and then being committed to the base. In general, we expect users to run network applications in IFS while mainly committing data files, so this is a worst case scenario for tainting and logging during normal user activity.

Tables 6 and 7 shows the tainting and logging results. Table 6 shows the number of days over which taint propagation was performed, the total amount of data and the number of system call events logged by Forensix (in million), the number of file namespace related system calls, and the total number of files that either existed on the system or were created during the 10 or 14 day tainting period. Table 7 shows several different client or server applications that we tainted (one at a time), the number of existing tainted files at the end of tainting period, and the number of tainted events that would have been logged by Solitude (also shown as a percentage of the total number of events logged by Forensix). These tainted events include the namespace events (tainted or otherwise) shown in Table 6 that are always logged in Solitude. Table 7 shows that the number of tainted files is relatively small for all applications, including applications like firefox that was run over 100 times during the tainting period, and hence post-intrusion base file-system recovery should be a feasible option. The percentage numbers for the applications also shows that the total amount of logging in Solitude should be much smaller than our original Taser/Forensix system.

6.4 Performance Overhead

We measured the overhead introduced by Solitude by running a set of benchmarks representing different client or server workloads. We ran two client workloads within an IFS: 1) untar of a Linux kernel source tarball, representing a filesystem-intensive workload, and 2) kernel build of the Linux sources, which is mainly CPU bound and determines the overhead imposed when running similar CPU bound applications in a regular desktop environment. We ran three server workloads in an IFS: 1) a large 230 MB file download, which stresses the file-system read performance and represents a media streaming server, 2) a large 230 MB file upload, which stresses the file-system write performance and represents an FTP or a video blogging site, and 3) the `Apache ab` benchmark, which stresses a standard Apache web server by issuing back-to-back requests with four concurrent processes running 20 clients that request files ranging from 1KB to 15KB, and is representative of a loaded server environment.

We ran the tests on a Solitude-enabled Ubuntu Linux 6.06 machine with four Intel(R) Xeon(TM) CPU 3.00GHz processors, 2GB of RAM and a local `ext3` hard disk. The client machine for the server experiments is connected to the target machine with a Gigabit network. We repeated each test at least 5 times and our results are averaged over these tests.

Figure 5 shows the performance overhead of Solitude for the five benchmarks compared to a regular Linux system. The y-axis shows the overhead in terms of running time for the first four experiments and in terms of network throughput for the CPU-saturated web server benchmark. The top graph shows the overhead for a process running in IFS and the bottom graph for a process running in base. Each segment of the bar shows the overhead introduced by the various components of Solitude. We obtained these results by starting with the base Linux system and then running experiments that progressively added these components one at a time. For an IFS process, these components include 1) the pass-through user-level file system built on FUSE, 2) the basic copy-on-write IFS environment, 3) IFS sharing and capability policy module, and 4) the kernel-level tainting module. The base components include the tainting module and the backend logging (and recovery) system. In both cases, the tainting module is run with no tainted files or processes to isolate the overhead introduced by logging. For the logging component, we taint the application and run the test to measure the overhead of running an entire application that was downloaded in an IFS and committed to the base.

The Untar test creates a large number files and directories, stressing the IFS file system. The FUSE overhead is largely a consequence of filesystem operations being redirected into user-space code which then makes more system calls into the kernel, and as a result, the user-level IFS code also incurs significant overhead. We expect both these overheads to decrease dramatically with a kernel-level implementation. The Solitude overhead occurs almost entirely due to hard links. Solitude, in addition to tainting, provides a file generation number for uniquely identifying files to the IFS code. As described in Section 5.1.1, the code stores the inode and generation number in a persistent mapping table for correctly handling the multiple names of a file due to hard links. In the future, we plan to assess whether hard links are sufficiently useful for IFS applications to justify the implementation complexity and overhead. For a base process, logging introduces significant overhead because all file and directory updates are logged to the backend system. This represents the worst case scenario when the entire `tar` application is tainted and run in the base.

The Build and the Apache benchmarks have smaller overhead than Untar in IFS and minimal overhead in base. The Upload

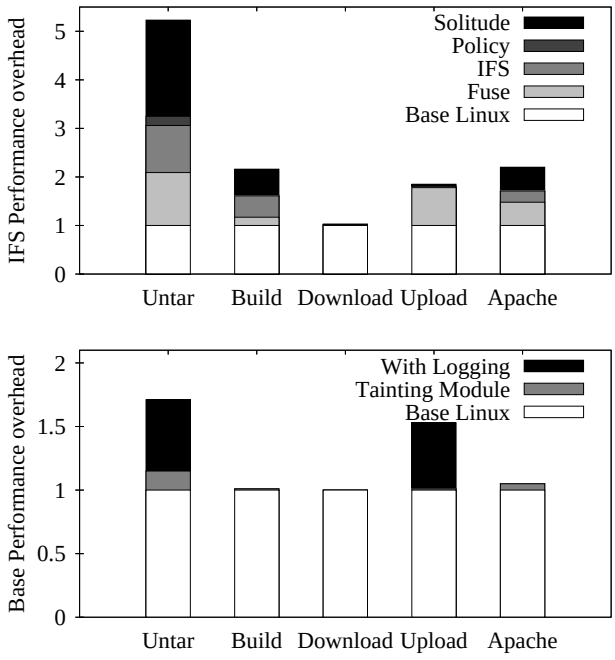

Figure 5: Performance overhead in IFS and base

benchmark stresses the FUSE code in IFS and the logging code in base since the tainted file is logged to the backend. The Download benchmark has no overhead.

7. RELATED WORK

There are several areas related to this work, access control, sandboxing techniques, file systems, and intrusion analysis and recovery. Access control policies restrict access to a system and its objects based on a set of discretionary or mandatory policies. Discretionary policies permit users to entirely determine the access that is granted to the resources they own, allowing them to give access to these resources to unauthorized users, whether by accident or malice. For example, in the Unix context, a file owner may set file permissions incorrectly, allowing implicit sharing of files with other users. Users can use Unix groups to share files, but these groups must be created by administrators, and it is not easy to provide per-application isolation with the Unix access control model. In contrast, IFS environments provide application isolation by default, and when applications do not require any Linux capabilities, the applications can be started in IFS environments by any user. Unix systems provide setuid capabilities that allow bypassing all access controls, making it dangerous for programs to possess these capabilities. IFS provides a finer-grained capability model that enhances Linux capabilities.

Mandatory policies enforce explicit sharing and are specified by an administrator based on the principle of least privilege. For example, SELinux [16] provides a powerful mandatory access control model, but it is commonly acknowledged that designing SELinux policies is a complicated process [10]. Solitude provides a simpler, but more coarse-grained isolation model in which policies are primarily needed for file sharing. However, more importantly, access control requires correctly specifying policies, while with our commit sharing policy, errors can be handled until commit is performed. Furthermore, Solitude audits and tracks commits and dependant changes so that if our sharing policies are used incorrectly and lead to attacks, it is still possible to recover the system.

The UMIP model, similar to Solitude, aims to preserve system integrity in the face of network-based attacks [14]. This model leverages information available in existing discretionary access control (DAC) policies to derive file labels for mandatory integrity protection. The basic UMIP policy partitions processes into low and high integrity. When a process performs an operation that potentially contaminates it, such as via reading from a network socket or communicating with another low integrity process, it drops integrity and cannot perform sensitive operations. The basic UMIP policy is enhanced with capability exceptions to support server applications. Our capability model was developed concurrently and has many similarities with UMIP capabilities. The primary difference is that our default policy is read sharing and not read deny, and hence our policy files are easier to specify because they typically do not need exceptions for reading files. More importantly, UMIP does not provide isolation to client-side applications run by the *same* user because it uses DAC policies to configure its policies. Since UMIP is an access control mechanism, it shares the limitation with SELinux that the policies must be correctly specified when files are updated. In contrast, our copy-on-write approach allows files to be in *both* low and high integrity states with explicit commits to raise the integrity of the files.

Information flow control systems employ the principle of least privilege to limit the impact of software vulnerabilities. For example, HiStar [38] allows users to specify precise security policies, but it often requires restructuring applications to meet its security goals. Our focus is on improving isolation by retrofitting existing applications.

Sandboxing techniques such as virtual-machine isolation and operating system-level virtualization ensure that the effects of an application are constrained within a restricted environment. Hypervisor-based virtual machines (VM) can provide strong isolation guarantees but they have limited support for sharing. For example, a virtual machine can be used to run multiple versions of the Office word processor, but each machine has its own separate desktop that may lead to a confusing and error-prone user experience.

A second virtualization approach that trades security for efficiency is to use operating system-level virtualization in which a single physical server is partitioned so that it appears as multiple servers that can be administered independently. This approach has been implemented in several operating systems such as BSD [11], Solaris [22] and Linux [28]. While similar to our isolation environment, OS virtualization is still designed primarily for isolating applications run by untrusting users (e.g., the different customers of a service provider) and thus focuses on avoiding denial-of-service attacks and provides limited sharing. For instance, in university or small corporate environments, a single machine is often used to run several server applications such as a web server, mail server, print server, etc. on behalf of the same set of users. With OS virtualization, by default, each of these server applications would require its own list of users and user directories. We envision using isolation environments for different applications run by the same user or by a group of users within the same organization and thus aim to provide better support for sharing and ease of use.

CapDesk [36] strives to enforce the principle of least authority by offering interactive policy configuration, such as granting access to files, when running applications. When trusted policy files are not available, a similar approach could be used in Solitude. Microsoft has recently released its Softgrid/SystemGuard technology for virtualizing applications [2]. Softgrid uses a single OS, but uses the SystemGuard virtual application environment to keep application

dependencies (DLLs, registry entries, fonts, etc.) separate from the rest of the system, which allows streaming and running multiple versions of an application such as Office within the same OS. SystemGuard uses a copy-on-write file system but does not allow users to commit applications or their configurations to the base, and also does not allow for auditing, tracking or recovery of the base system. GreenBorder is another application virtualization technology that provides copy-on-write protection, but is tailored to provide protection for specific applications such as web browsers [8].

In One-way Isolation [30], untrusted processes observe the environment of their host system, but the effects of these processes are isolated from other applications. Once the code is trusted, all changes made by it can be committed to the host system. Our commit sharing method is motivated by this work. However, while this work proposes using one-way isolation for testing and debugging, we propose to limit sharing by running applications in the long term in this environment. As a result, our system provides support for explicit sharing across the isolation environments, an enhanced capability model for running server applications securely, the ability to commit selectively as well as perform recovery even after data is committed.

The idea of per-process namespaces first appeared in Plan 9 [21], although it was largely motivated by representing various resources as file systems. Solitude uses namespaces to isolate changes made by each application. Solitude has similarities with the Ventana file system [20] that provides sharing and file-level rollback with a rich file-system level versioning scheme. However, Ventana primarily focuses on using and managing virtual disks in a virtual machine environment, while Solitude aims to maintain integrity and provide recovery facilities after an attack. Many file systems [25, 19] have been developed for creating snapshots for versioning and recovery. These file systems typically implement versioning at the block level which is simpler to implement and provides good performance. However, our goal is to enable limited sharing and selective commits at the file-system level, and hence IFS uses copy-on-write at the file-system level.

Transactional file systems, for example QuickSilver [7] and Vista's TxF (transactional file system) [34], allow file system operations to be handled like transactions so that all the changes within a transaction are committed to disk atomically and the intermediate states of a transaction are not visible to other applications or transactions within the same application. Both file systems require changes to applications to use a transactional interface to start, abort or commit a transaction, and they use a pessimistic locking mechanism for ensuring consistency. Quicksilver holds read locks on files until the file is closed and write locks until the end of a transaction. Directories are locked when they are modified, for example when a directory is renamed, created or deleted. TxF's locking mechanism is also very similar to QuickSilver. However, a file can be read and written in two different transactions concurrently. In this case, the reads do no see the modifications made by the other transaction.

In contrast to QuickSilver and TxF, our IFS environment supports existing applications without requiring any changes to these applications. It provides transactional semantics via commit sharing at the IFS granularity, and hence transactions can exist for long periods of time. To ensure availability in the face of long-running transactions, IFS uses an optimistic concurrency control method that allows the different IFS environments to concurrently access and modify files. IFS transactions can either be rolled back by discarding the entire IFS environment or IFS allows using resolution policies when conflicts occur during a commit [32].

File-system workloads have been extensively characterized for improving file system performance through prefetching and caching [35, 23]. We observe that sharing of files between different applications is relatively uncommon and thus suggest using explicit file-sharing mechanisms.

Several efforts have focused on analysis and recovery of compromised systems. The Repairable File Service [39] logs file system activity and performs contamination analysis to provide system recovery after an intrusion. Backtracking [12] helps determine the source of attacks by tracking dependencies among kernel objects in reverse time order. Taser [6] determines and reverts the effects of malicious file-system activities by tracking similar dependencies in reverse and forward order. Hsu et al. [9] propose a malware removal framework that allows rolling back untrusted updates.

8. CONCLUSIONS

We have described Solitude, a multiple namespace file-system isolation environment designed for existing applications. Solitude explores how the benefits of a shared namespace may be preserved while limiting the implicit sharing that allows compromises to propagate and confounds forensic analysis and recovery. To enable sharing, Solitude requires an explicit file sharing specification between its isolation environments and the base file system, which also helps improve accountability of changes to persistent state. Solitude's capability restrictions ensure that even if malware compromises a legitimate program running with some privileges in its isolation environment, then it would be unable to embed itself deep into the system (e.g. by loading a kernel module) because the host application would likely possess only a few capabilities. When sharing operations lead to an attack in the base environment, Solitude provides the ability to perform system-level intrusion analysis and file-system recovery based on tracking contamination.

We are currently exploring several avenues of future work. We plan to study whether our isolation and recovery methods are applicable to other systems such as Windows that provide several means of communication between applications including registry entries. We are investigating the use of handler programs that are activated on sharing or commit operations as a way of detecting malicious synchronization operations, For example, a spyware detector could be run during each commit. Currently, IFS environments need to be started manually. We plan to address these issues with a kernel-level IFS implementation. Finally, we plan to explore the use of the Solitude infrastructure as a debugging environment and for configuration management.

Acknowledgments

The ideas in this paper were refined during several discussions with Andrew Warfield. We greatly appreciate the valuable and detailed feedback received from the anonymous reviewers and from our shepherd, Gernot Heiser. We wish to thank Alex Varshavsky, Eyal de Lara, Stefan Saroiu and several other members of the SSRG group in Toronto who provided comments on initial drafts of the paper.

9. REFERENCES

[1] Brian Cornell, Peter Dinda, and Fabián Bustamante. Wayback: A user-level versioning file system for linux. In *Proceedings of the USENIX Technical Conference*, pages 19–28, June 2004.

[2] Microsoft Corporatin. Microsoft SoftGrid. `http://www.microsoft.com/systemcenter/softgrid/evaluation/virtualization.mspx`, 2007.

[3] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 133–147, 2005.

[4] Steve Friedl. Best practices for UNIX chroot() operations. http://www.unixwiz.net/techtips/chroot-practices.html, January 2002.

[5] Ashvin Goel, Wu chang Feng, Wu chi Feng, David Maier, and Jim Snow. Automatic high-performance reconstruction and recovery. *Journal of Computer Networks*, 51(5):1361–1377, April 2007. From Intrusion Detection to Self-Protection.

[6] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2005.

[7] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82 – 108, 1988.

[8] Matt Hines. Google buys into security, acquires GreenBorder. http://www.infoworld.com/article/07/05/29/Google-buys-into-AV_1.html, May 2007.

[9] Francis Hsu, Hao Chen, Thomas Ristenpart, Jason Li, and Zhendong Su. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the Annual Computer Security Applications Conference*, December 2006.

[10] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the USENIX Security Symposium*, pages 59–74, August 2003.

[11] Poul-Henning Kamp and R.N.M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the Second International SANE Conference*, 2002.

[12] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 223–236, October 2003.

[13] John Leyden. Spyware poses as Firefox extension. urlhttp://www.theregister.co.uk/2006/07/26/firefox_malware_extension, July 2006.

[14] Ninghui Li, Ziqing Mao, and Hong Chen. Usable mandatory integrity protection for operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–178, 2007.

[15] Linux. Man capabilities(7) in Linux man page. Confirming to POSIX.1e.

[16] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the Freenix Track of USENIX Technical Conference*, June 2001.

[17] Bharat Mediratta. Gallery photo album organizer. http://gallery.menalto.com/, 2004.

[18] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, February 2005.

[19] Zachary N.J. Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, May 2005.

[20] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the Networked Systems Design and Implementation (NSDI)*, May 2006.

[21] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in Plan 9. *ACM Operating Systems Review*, 27(2):72–76, 1993.

[22] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the USENIX Large Installation Systems Administration Conference*, 2004.

[23] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of the USENIX Technical Conference*, June 2000.

[24] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[25] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 110–123, December 1999.

[26] Secure chroot barrier - Linux-Vserver. http://linux-vserver.org/Secure_chroot_Barrier, viewed in Aug 2007.

[27] Fareha Shafique. Application-level file system isolation. Master's thesis, University of Toronto, December 2007.

[28] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the EuroSys conference*, pages 275–287, 2007.

[29] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[30] Weiqing Sun, Zhenkai Liang, R. Sekar, and V.N. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Proceedings of the Network and Distributed System Security Symposium*, February 2005.

[31] Miklos Szeredi. File system in user space (FUSE). http://fuse.sourgeforge.net.

[32] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP)*, pages 172–183, December 1995.

[33] David Thiel. Exposing vulnerabilities in media software. Black Hat USA 2007, http://www.blackhat.com/html/bh-usa-07/bh-usa-07-speakers.html#thiel, August 2007.

[34] Surendra Verma and Charles Torre. Vista transactional file system, December 2005. http://channel9.msdn.com/Showpost.aspx?postid=142120.

[35] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, December 1999.

[36] David Wagner and Dean Tribble. A security architecture of the combex darpabrowser architecture, March 2002. http://www.combex.com/papers/darpa-review/security-review.pdf.

[37] Andy Watson and Paul Benn. Multiprotocol Data Access: NFS, CIFS, and HTTP. Technical Report TR3014, Network Appliance, Inc., 1999. http://www.netapp.com/tech_library/3014.html.

[38] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, November 2006.

[39] Ningning Zhu and Tzi-Cker Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the IEEE Dependable Systems and Networks*, pages 217–226, June 2003.